Konwersje liczb całkowitych bez znaku do postaci ciągów tekstowych (system szesnastkowy i dziesiętny)

Zarówno dane (niezależnie od ich typu), jak i instrukcje procesora przechowywane są w pamięci w postaci liczb. Ciągi „0" i „1" reprezentacji binarnej tych liczb odpowiadają ustawionym (bądź skasowanym) przerzutnikom pamięci statycznej lub zgromadzonemu ładunkowi w kondensatorach komórek pamięci dynamicznej.

Aby wydrukować dane z pamięci komputera potrzebna jest procedura konwersji z postaci binarnej na formę bardziej zrozumiałą dla człowieka.

Konwersja liczby z systemu binarnego na szesnastkowy jest bardzo prosta (podstawy obu systemów są potęgami dwójki). Dodatkową zaletą systemu szesnastkowego jest wykorzystanie tylko dwóch cyfr do zapisu wartości zmiennej jednobajtowej bez znaku (00 – FF), podczas gdy w systemie dziesiętnym, w ogólnym przypadku, trzeba użyć trzech cyfr (000 – 255).

„Ciąg" binarny wystarczy podzielić (od prawej – najmłodszej strony) na czterobitowe grupy (*półbajty – nibbles*), a następnie każdej z nich przyporządkować symbol odpowiadający cyfrze systemu szesnastkowego.

Np. 14–bitowe „słowo" 11010111001111_{BIN} można „podzielić" na czterobitowe części:
0011_{BIN} 0101_{BIN} 1100_{BIN} 1111_{BIN} odpowiadające liczbie: $35CF_{HEX}$ w systemie szesnastkowym.

W praktycznym „algorytmie" konwersji należy uwzględnić kolejność znaków odpowiadających cyfrom 0 – 9 oraz literom A – F w tablicy ASCII. Cyfrom 0 – 9 odpowiadają znaki o numerach 48 – 57.
Kolejne 7 znaków (: ; < …) należy pominąć (58 – 64). Cyfrom systemu szesnastkowego (A – F) przyporządkowane zostały kody od 65 do 70.

Przykładowe pseudokody różnych „algorytmów" konwersji liczba -> znak (znaki) ASCII.

1. Konwersja półbajtu.

Zał.: argument przekazany w czterech młodszych bitach rejestru *%al*, cztery starsze bity są wyzerowane.
Zwracana wartość: numer znaku ASCII (8–bit): również w rej. *%al*.

```
convert_nibble:

  if (%al < 10) %al += 48 else %al += 55

  return %al
```

Oczywiście nie jest to jedyny sposób – patrz p. 5.

2. Konwersja bajtu wykorzystująca *convert_nibble*, nieoptymalna:

Zał.: argumenty: liczba-bajt w rejestrze *%al*.
Zwracana wartość: dwa znaki ciągu tekstowego ASCII w *%ax*. Porządek bajtów *big endian*.: w *%al* starszy bajt, w *%ah* – młodszy.

```
convert_byte:

  temp = %al      //kopia wartości %al; %al będzie nadpisywany
  %al &= 0x0F     //pozostaw tylko 4 młodsze bity w %al

  convert_nibble  //konwersja młodszego półbajtu %al

  %ah = %al       //wynik konwersji (bajt w %al) do %ah
  %al = temp
  %al >>= 4       //przesuń starszy półbajt na miejsce młodszego (4 bity w prawo)
  convert_nibble  //konwersja starszej połówki bajtu

  return %ax
```

3. Konwersja wielobajtowych typów danych.

Zał.: argumenty: liczba w rejestrze „*%a”* (np. *%rax* odpowiedniej długości), adres zapisu w *%rdi* (zaczynając od najmłodszej pozycji), rozmiar zmiennej podlegającej konwersji (w bajtach: 2, 4 lub 8) w *%ecx*. Funkcja wykorzystuje poprzednią procedurę *convert byte* (i pośrednio *convert_nibble*). Wersja nieoptymalna.

```
convert:

  for (%ecx=2/4/8; %ecx>0; %ecx--){

  temp’ = %a        //kopia zawartości rejestru %a
  convert_byte      //wywołaj konwersje najmłodszego bajtu

  @(%rdi) = %ax    //zapisz oba znaki: starszy w %al, młodszy w %ah
                   //odpowiednio pod adresy: %rdi i %rdi+1
                   //zapis odbywa sie w porządku bajtów little endian!

  temp’ >>= 8       //ustaw kolejny bajt do konwersji na miejscu najmłodszego
  %a = temp’        //przygotuj argument dla kolejnego wywołania convert_byte
  %rdi -= 2         //przesuń wskaźnik miejsca zapisu o dwa bajty:
                    //od pozycji najmłodszych w kierunku starszych.
  }

  return –
```

Jako *temp* w p. 2. i 3. mogą służyć dowolne nieużywane (również w funkcjach *convert_byte* i *convert_nibble*!) rejestry ogólnego przeznaczenia (odpowiedniego rozmiaru). Ew. można zaalokować odpowiedni obszar pamięci.

Uwaga: kod powyższych funkcji można uprościć/zoptymalizować np. wykorzystując licznik pętli (*%ecx*) do adresowania miejsca zapisu (patrz p. 6.) oraz rozkaz *xchg* (*swap*)do ulokowania obu znaków reprezentujących bajt w odpowiednim porządku w rejestrze *%ax*. Zamiast ustawionej stałej liczby iteracji, obliczenia można powtarzać dopóki wartość w *%a* jest niezerowa.

4. Konwersja wielobajtowej liczby na system dziesiętny:

… generalnie jest również prosta, przedstawiony tutaj „algorytm” wymaga jednak obliczania reszty z dzielenia (mod 10).

Operacja dzielenia jest stosunkowo skomplikowana (w porównaniu np. do dodawania) i większość prostych mikrokontrolerów 8–bitowych (8048, MC6805) i procesorów (np. MC6800, 8085/Z80) jej sprzętowo nie wykonuje. Należy wtedy napisać dodatkową funkcję. W architekturze x86 modulo obliczane jest podczas wykonywania dzielenia.

Proszę zwrócić uwagę na składnię instrukcji *div* (dzielenie bez znaku) i miejsce zapisywanych wyników. Np. wersja 32–bitowa rozkazu *div* dzieli 64–bitową dzielną zapisaną w parze rejestrów: *%edx* : *%eax* (*high* : *low*) przez wartość 32–bitowego dzielnika, będącego **jedynym argumentem** tego rozkazu (rejestru lub zawartości pamięci). W wyniku operacji, 32–bitowy iloraz zapisywany jest w *%eax*, a 32–bitowe modulo w *%edx*.

Ponieważ w programie używamy 32–bitowej dzielnej (np. typ *unsigned int*), przed wykonaniem *div* rejestr *%edx* należy wyzerować. Jeśli się o tym zapomni, wynik może być błędny. W przypadku gdy obliczony iloraz przekroczy 2^{32}-1 (maks. *unsigned int*) zgłoszony zostanie wyjątek – w Linuxie: błąd obliczeń… zmiennoprzecinkowych (sic!).

Analogicznie sprawa dzielenia bez znaku wygląda z użyciem rejestrów 16– i 64–bitowych. Dzielenie „ośmiobitowe” wymaga jednak umieszczenia dzielnej w rejestrze *%ax* (tj. parze *%ah* : *%al*), dzielnika w ośmiobitowym argumencie rozkazu, wynik zwracany jest w: *%ah* – modulo i w *%al* – iloraz.

Zał.: jak poprzednio: liczba w *%a*, adres zapisu (zaczynamy najmłodszej pozycji – jedności) w *%rdi*:

```
convert2dec:

  do {
    %q = %a  /  10        //%q - iloraz
    %m = %a mod 10        //%m – modulo 10 – wartość: od 0 do 9
    %m |= 48              //przesunięcie ASCII
    @(%rdi) = %m          //zapisz znak
    %rdi--                //przesuń wskaźnik zapisu
    %a = %q
  } while (%a != 0)       //powtarzaj dopóki jest co dzielić
```

5. LookUp Table (LUT)

Inną metodą konwersji, eliminującą konieczność wykonywania większości obliczeń (za wyjątkiem np. obliczania adresów) jest wykorzystanie stablicowanych, gotowych elementów wyjściowego ciągu tekstowego.

Idea jest bardzo prosta:

wartość argumentu funkcji jest jednocześnie numerem (indeksem) elementu tablicy (*LookUp Table – LUT*), w którym to elemencie przechowywana jest żądana informacja wyjściowa – zwracana przez funkcję.

„Idąc dalej" – na podstawie jednego znaku ASCII (bajtu) sterownik wyświetlacza (albo drukarki tekstowej) wybiera z kolejnej tablicy - generatora znaków - jego graficzną reprezentację: font. W trybie graficznym (np. 320 x 200 x 256 kolorów) karta VGA na podstawie jednej „wartości" piksela (bajt) odczytuje z wcześniej zaprogramowanej, większej palety, wartości trzech składowych koloru R, G, B itp., itd.

Tym sposobem można również zastąpić obliczanie wartości funkcji np. sin – odczytem „gotowej" wartości z tablicy, której element wybiera argument stablicowanej funkcji. Oczywiście jest to reprezentacja dyskretna – dokładne wartości funkcji są wyznaczone tylko dla zbioru konkretnych argumentów, w pewnym przedziale.

Wadą tablicowania może być (nie musi!) sama tablica, zajmująca pewien obszar pamięci (tutaj: pamięci RAM – danych, w mikrokontrolerze np. fragment nieulotnej pamięci programu) oraz czas dodatkowego dostępu do pamięci danych. Wszystko zależy od konkretnego zastosowania i przyjętego kryterium optymalizacji (rozmiar kodu, rozmiar danych, szybkość wykonania kodu, rozdzielczość – dyskretyzacja tablicy…).

Np. wartość półbajtu (od 0 do 15) adresuje jeden z szesnastu (2^4) elementów tablicy, spod którego odczytywany jest znak kodowany zgodnie tablicą ASCII. Tablica zajmuje obszar szesnastu bajtów i przykładowo można ją zapisać jako:

```
lut8:  .ascii "0", "1", ... ,"9", "A", ... , "E", "F"
```

natomiast prosty odpowiednik procedury *convert_byte* (założenia jak w punkcie 2.) wykorzystujący LUT może mieć postać:

```
convert_byte:

  %dl = %al               //kopia argumentu do %dl
  %eax &= 0x0F            //zostaw tylko 4 młodsze bity (0 – 15) w %eax
                          //do adresowania używamy rej. 32/64-bitowych
  %ah = lut8[%eax]        //odczytaj młodszy znak z tablicy do %ah

  %dl >>=4                //starsza połówka bajtu będzie adresować tablice
  %edx &= 0x0F            //jw.
  %al = lut8[%edx]        //odczytaj starszy znak z tablicy do %al

  return %ax              //wynik konwersji w %ax, kolejność big endian
```

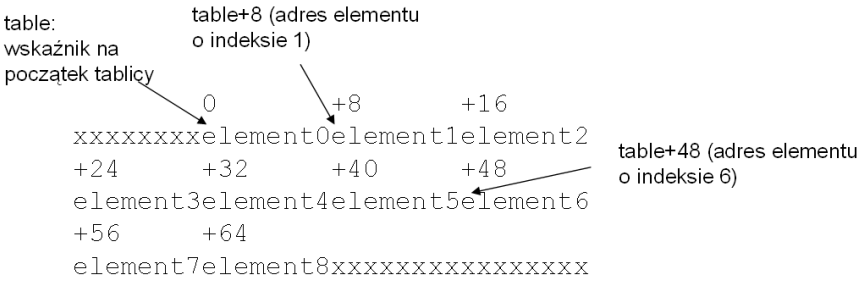
W powyższym przypadku wyeliminowane zostało wywoływanie „*convert_nibble*” oraz sprawdzanie podzakresu, w którym mieści się argument funkcji (0 – 9 czy 10 – 15). Usunięcie porównań i skoków ułatwia i przyspiesza wykonanie kodu przez procesor potokowy. Dwa dostępy do tablicy (pamięci) są rekompensowane brakiem operacji na stosie związanych z odkładaniem i ściąganiem adresu powrotu z *convert nibble*.

Tablicę można również zbudować z 256 elementów dwuznakowych – tak, aby pozwalała „w locie” zamienić cały bajt na fragment wyjściowego ciągu tekstowego. Kosztem większej tablicy – jeszcze bardziej uprości się sama procedura i skróci czas konwersji.

6. Adresowanie tablic

Ułożenie w pamięci kolejnych 8–bajtowych elementów przedstawia się następująco:

```
table: .quad element0, element1, element2, element3, element4, ...
```



Czyli: adres *i*-tego elementu (licząc od zera) = adres_początku_tablicy + indeks * rozmiar_elementu (w bajtach).

Pełny tryb adresowania komórek pamięci przez procesor zgodny z x86 dopuszcza użycie dwóch rejestrów oraz stałej liczby (składnia AT&T):

```
mov  stała(%rejestr1, %rejestr2, skala) , %rejestr3                    (*)

adres komórki = stała + %rejestr1 + %rejestr2 * skala                 (**)
```

gdzie:

%rejestr2 – najczęściej zawiera indeks – numer elementu tablicy liczony od zera skalowany przez rozmiar elementu *skala*,

skala – stała–potęga dwójki (1, 2, 4, 8) – zależy od rozmiaru (w bajtach) elementów tablicy (mnożenie wewnętrznie jest zastępowane przesunięciem bitowym w lewo),

%rejestr1 – może być przydatny przy adresowaniu np. obszarów alokowanych dynamicznie, tablic dwuwymiarowych itp.,

stała – np. etykieta tablicy = adres pierwszego (indeks = 0) elementu statycznego bloku danych,

%rejestr3 – miejsce docelowe, będzie zawierał dane rozpoczynające się od obliczonego (**) adresu.

Uwaga 1. Podczas adresowania tablic nie muszą być wykorzystywane wszystkie powyższe elementy: np. odczyt wartości elementu o indeksie w *%rax*, z jednowymiarowej, statycznej tablicy *table* złożonej z elementów 64–bitowych ma postać:

```
mov table( , %rax , 8) , %rdx        # %rdx = dane (8 kolejnych bajtów) spod adresu: table+%rax*8
```

Uwaga 2. Z pamięci zostaje odczytana (lub zapisana – po zamianie operandów miejscami) liczba bajtów odpowiadająca rozmiarowi *%rejestru3* (*), np. *%eax* – 4 bajty, *%ax* – 2 bajty.

W niektórych przypadkach (np. zapis do pamięci stałej wartości - możliwe kodowanie w 1, 2, 4 lub 8 bajtach), aby jednoznacznie określić typ danych należy dodać do instrukcji odpowiedni sufiks:
q – 64, **l** – 32, **w** – 16, **b** – 8 bitów.

`mov`**`b`** `$3 , etykieta` – oznacza zapis jednego bajtu: 00000011 pod wskazany adres-etykietę
`mov`**`w`** `$3 , etykieta` – zapis słowa: 00000000 00000011 zaczynając od adresu *etykieta* (w x86 – porządek bajtów *little endian*).

Adres w instrukcji np. `mov $3 , etykieta` sam w sobie nie zawiera informacji o typie danych (liczbie zapisywanych bajtów), więc kompilator *as* zgłosi błąd podczas tworzenia pliku pośredniego (*.o*).

Uwaga 3. W architekturze x86 generalnie tylko jeden operand instrukcji może odnosić się do lokalizacji w pamięci (tzn. nie można wykonać rozkazów typu: `mov pamiec1 , pamiec2 ; add pamiec1 , pamiec2`).

Uwaga 4. Procesor pracuje w trybie *long mode* - do adresowania komórek pamięci używa się tylko rejestrów 32- lub 64-bitowych.

7. Przykładowe zadania na laboratoria 3. i 4.

1. W pliku *hex2str.s* dopisać w asemblerze funkcje:

– *convert_nibble*,
– *convert_byte*,
– *convert* (dla typów wielobajtowych).

2. Sprawdzić poprawność konwersji różnych typów całkowitoliczbowych (bez znaku).

3. Zmienić „algorytm” konwersji półbajtów na wykorzystujący tablicę LUT – jednoznakową.

4. Napisać funkcję pozwalającą konwertować w jednym kroku cały bajt na odpowiadający jego wartości (*hex*) dwuelementowy ciąg tekstowy przy użyciu tablicy z pliku *lut.s*. Zoptymalizować procedurę konwersji liczb wielobajtowych.

– przeanalizować zalety i wady, zyski i straty w rozwiązaniach z zad. 3 i 4.

5. Uzupełnić plik *dec2str* o funkcję do konwersji liczby na ciąg tekstowy reprezentujący jej wartość w systemie dziesiętnym.